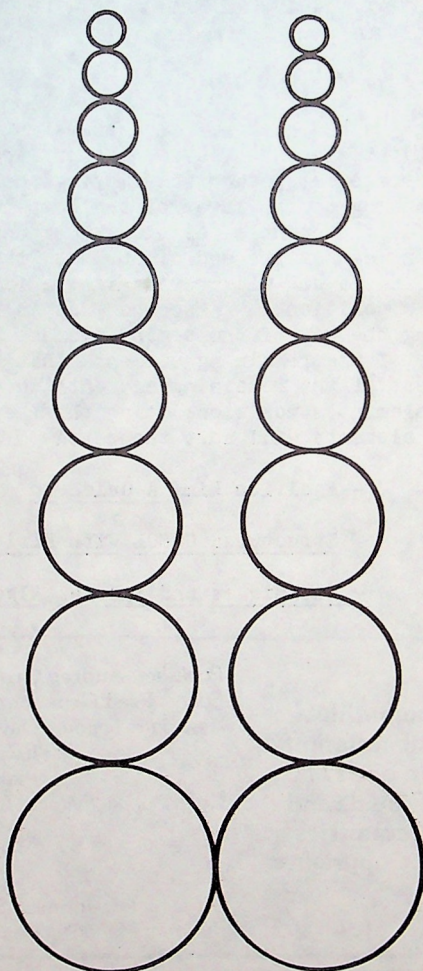


# Popular Computing

Volume 8 Number 11

November 1980

92



Bubble  
Sorting



# Bubble Sorting

Bubble sorting is a very popular subject, to judge by its appearance in nearly every introductory or language textbook. It is known also under the names interchange sorting, exchange sorting, and sinking sort.

It is one of those concepts that everyone knows; that is, everyone who has ever coded a bubble sort feels that his is the normal version and, moreover, that everyone else understands this fact. But no two texts seem to agree on just what bubble sorting is, and many texts describe sorting schemes that are far removed from what most people call bubble sorting. Just as an example:

"The bubble sort works by comparing the key of the first element of a set against the keys of all other elements, each time moving the element corresponding to the lower of the compared keys to the first element's position. When the operation reaches the end of the set, the lowest element in the set will reside in the first element's position. A second pass is made, comparing the key of the second element against the keys of the remaining elements that moves the lowest of the remaining elements to the second element's position; doing this repeatedly for all elements will sort the set."

--Pacífico Lim, A Guide to

Structured COBOL With Efficiency

Techniques and Special Algorithms.

POPULAR COMPUTING is published monthly at Box 272, Calabasas, California 91302. Subscription rate in the United States is \$20.50 per year, or \$17.50 if remittance accompanies the order. For Canada and Mexico, add \$1.50 per year. For all other countries, add \$3.50 per year. Back issues \$2.50 each. Copyright 1980 by POPULAR COMPUTING.

@ 2023 This work is licensed under CC BY-NC-SA 4.0

*Publisher:* Audrey Gruenberger

*Editor:* Fred Gruenberger

*Associate Editors:* David Babcock

Irwin Greenwald

Patrick Hall

*Contributing Editors:* Richard Andree

William C. McGee

Thomas R. Parkin

Edward Ryan

*Art Director:* John G. Scott

*Business Manager:* Ben Moore

Another fairly popular text has this:

Bubble Sort

This method requires fewer steps if the data is partially in order. We search down the array  $B(I)$ , starting from  $I = 1$  and continuing until we find an element out of order. Thus if

$$B(1) \geq B(2) \geq \dots \geq B(K),$$

but

$$B(K) < B(K+1),$$

the  $(K+1)$ th element is the first element out of order. We then search back from  $B(K)$ , moving elements down one slot until we find the right place to put  $B(K+1)$ .

--C. W. Gear, Introduction to  
Computer Science.

There seems to be a consensus that the essence of bubble sorting, in any of its variations, is the interchange of two adjacent elements. So whatever technique Gear is describing, it is not what is generally known as bubble sorting.

The authority on sorting, of course, is Professor Donald Knuth's The Art of Computer Programming, Vol. 3. But Prof. Knuth omits bubble sorting completely, on the grounds, perhaps, that it is beneath contempt and so inefficient as to be suppressed.

This won't do; you can't kill an idea by ignoring it. Since the scheme appears in countless books, it is being taught--this month--to countless beginners. In many instances it is the only sorting scheme that those beginners see; thus, it will not only be perpetuated, but will be widely used when it shouldn't be.



Well, when should it be used? In its fundamental form, it has the tremendous virtues of being easy to code in any language, of being easy to debug and test, and of being almost idiot-proof; that is, it can be depended on not to spring any surprises on its user. I would venture a guess that right now more things are being sorted by one of the variations of bubble sorting than by all other schemes combined.

We might list some of those other schemes, to insure that we are communicating.

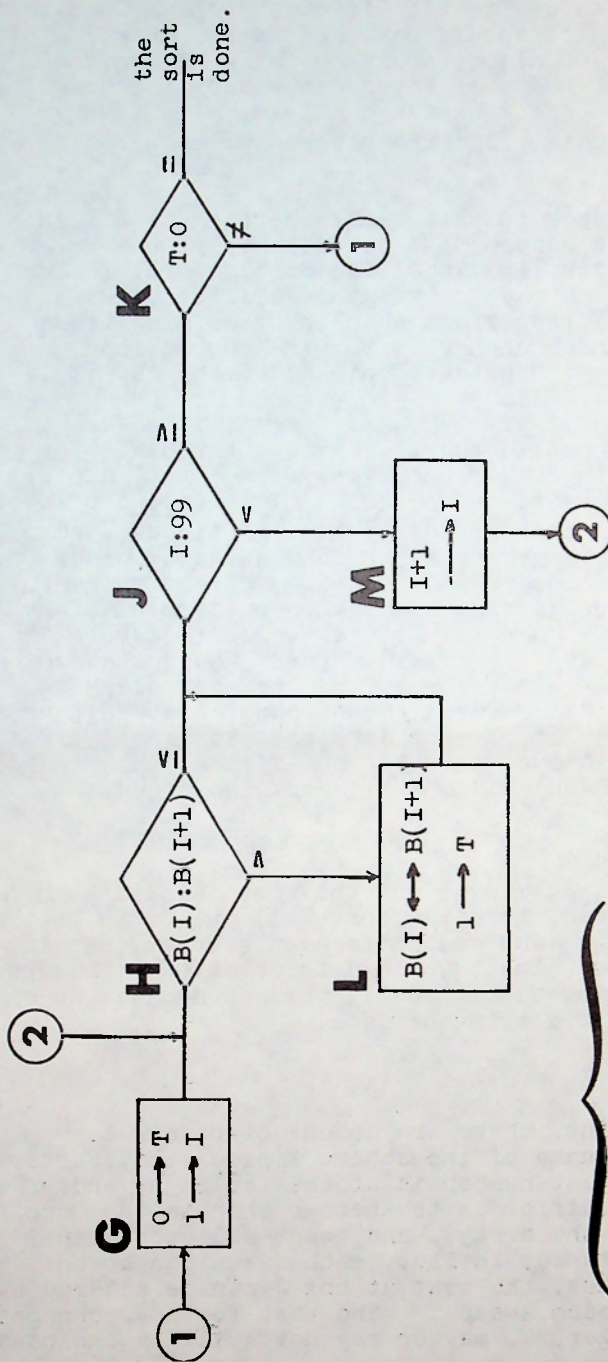
1. Direct internal sorting. These methods apply only to the situation where a few--say, 3, 4, or 5--items are to be sorted. See the article "Eight Is Not Enough" by Associate Editor David Babcock in our issue number 73 for a thorough treatment of this limited topic.

2. Repetitive internal sorting. This category (for which all the items to be sorted must be in central storage at once) includes Shell sorting (see our issue number 58 for a discussion of this technique), Quicksort, and Heap-sort.

3. External sorting. In these algorithms, the data to be sorted exists primarily on secondary media such as tapes, disks, or drums. The commonest scheme here is merge sorting, in which short strings are merged into longer and longer strings.

Now, a programmer has all these methods at his disposal. There are constraints that limit his choice. For example, internal sorting schemes must be rejected if there is insufficient room in central storage to hold all the data at once. Or, a multi-way merge cannot be used if there are not enough external storage devices available.

Suppose that the number of items to be sorted,  $K$ , is small, and the number of times that the sort is to be executed,  $N$ , is also small. By "small," let's say that the product of  $N$  and  $K$  is 5000 or less. Suppose that a programmer feels comfortable with either bubble sorting or Shell sorting. If he chooses Shell sorting, then the algorithm must be looked up (it is not the sort of thing that one codes from memory) and the program will have to be both debugged and carefully tested--the method is tricky. The elapsed time involved in choosing Shell sorting may outweigh by an enormous factor the wasted machine time involved in choosing bubble sorting in the first place.



Fundamental scheme for  
bubble-sorting an array  
of 100 items,  
 $B(1), B(2), \dots, B(100)$

Trigger T notes whether any  
interchange has taken place.



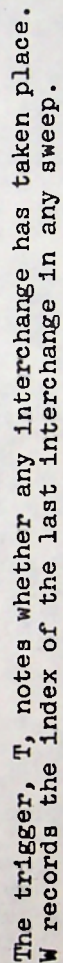
But the real-life situation is even simpler than that. No one ever asks how it should have been done, so the programmer's choice of algorithm is normally hidden and unquestioned. At that, if anyone does question the running time, then a quick switch can be made to a more efficient sorting scheme (and the programmer acquires a Hero medal).

But let's go back to what bubble sorting is all about. There is one sorting scheme that is definitely more inefficient, but which gets labelled "bubble sorting" in some texts: Scan the array to find the smallest value and exchange it for the first item; scan the remaining items to find the smallest and exchange it for the second item; and do this  $(K-1)$  times. Whatever this algorithm is, it is not bubble sorting.

What might be considered the basic interchange algorithm is given in the accompanying flowchart, made up to deal with 100 items to be sorted. In each sweep across the data array, each pair of adjacent items is compared and, if necessary, interchanged to establish ascending order for that pair. If an interchange is made, then a trigger,  $T$ , is set to one (the trigger having been set to zero before any sweep begins). When the trigger is still zero at the end of a sweep, the sort is complete (and the last sweep was a sequence check). The scheme tends to capitalize on any ordering that exists; for a set of data already in order, just one sweep is needed. For data that is exactly in reverse order, the scheme calls for  $K(K-1)(1/2)$  interchanges (box L on the flowchart) and  $K(K-1)$  comparisons (box H).

A simpler version leaves out the test involving the trigger,  $T$ . Instead, it calls for  $(K-1)$  sweeps to be made in every case. The coding is even simpler than in the fundamental case (simply arrange to count  $(K-1)$  sweeps at box K), but the method as usually presented takes no account of existing order in the data.

From that point, there are dozens of possible variations. The name of the scheme implies that, after one sweep, the largest number is at the far right end of the array (authors differ as to whether this is the "top" or the "bottom" of the array), and hence all subsequent sweeps can be shortened; in fact, each sweep can be one item shorter. Thus, the test at box J can be altered by one at the end of each sweep. The test for interchanges involving the trigger,  $T$ , may or may not still be included.





The results of some timing runs on six of these schemes are given in a chart (in which all times are in seconds). Each type was coded in floating point BASIC and timed by repeated runs. Like the EPA ratings of possible automobile mileage, the figures in the chart are only relative. Time taken to generate the data, for example, is longer for lines E and F than for lines C and D, but this difference is not reflected in the tabulated results.

It can be seen that for numbers already in sequence, there is no significant difference between the types, except, of course, for the one called SIMPLE. The best trade-off between ease of coding and efficiency of running time is to be found with type 6 (using the fundamental scheme found in the flowchart, but ending each sweep at the point of the last interchange of the previous sweep).

Or, the sweeps can alternate direction, thus also bubbling the smallest item to the far left end of the array. This notion can be combined with the trigger and with the idea of contracting each sweep; we thus have four more possible schemes.

Finally, it is noted that if the address of the last interchange is noted at box L, then the test at box J on the next sweep need proceed no further. And, of course, that improvement could be incorporated with any of the others.

A separate flowchart for type 6 is included.

The net result is this: there are at least a dozen different algorithms that can all be called BUBBLE SORTING. The more complex versions will save some machine time, but always at a cost of more code to write, debug, test, and execute. As has been pointed out many times, a crude but simple tool should probably not be tinkered with to sharpen it, unless the tinkering is itself simple (and obvious). Bubble sorting has its place--and is widely used--but the experienced programmer will know when NOT to use it.



	1	2	3	4	5	6
	SIMPLE	FUNDAMENTAL	Shorten each sweep by one element.	Alternate directions.	Alternate; shorten by one at each end of each sweep.	End each sweep at the last sweep's last interchange.
			(But terminate when there are no more interchanges)			
100 numbers already in sequence. <b>A</b>	162	2.45	2.05	3.16	3.08	2.90
200 numbers already in sequence. <b>B</b>	685	5.25	5.13	5.47	5.08	4.90
100 numbers in descending order. <b>C</b>	233	242	230	247	244	166.3
200 numbers in descending order. <b>D</b>	976	1006	925	404	796	659.5
100 numbers in random order. <b>E</b>	613	188	186	137	136.6	126
200 numbers in random order. <b>F</b>	840	818	725	549.3	650	495

Further research into the topic of bubble sorting brings out these facts:

Of 10 introductory texts surveyed, those that described the interchange operation itself did it exclusively in terms of:


```
Move A to TEMP
Move B to A
Move TEMP to B
```

and did not even hint at some of the other ways that this operation could be performed (as in our issue 56, page 13).

In the 10 texts (all of them quite recent), two described Type 1 (SIMPLE); three described Type 2 (FUNDAMENTAL); five described Type 3 (Fundamental, but shortening each left-to-right sweep); and two went to Type 6 (ending each sweep at the last interchange of the previous sweep).

Lim's book says "...the bubble sort is the most useful algorithm; it is one of the easiest sort algorithms to understand and is practically as efficient as other sort algorithms if sorting only a small set of data."

Wilfred Rule, in his Fortran text, says "The 'interchange' sort has been chosen for this discussion because it requires a minimum amount of additional internal storage and represents a good compromise between efficiency and simplicity."



```
1000 W = 99
1010 T = 0
1020 Q = W

1200 FOR I = 1 TO Q
1210 IF B(I) <= B(I+1) THEN 1300
1220 TT = B(I)
1230 B(I) = B(I+1)
1240 B(I+1) = TT
1250 T = 1
1260 W = I
1270 NEXT I

1300 IF I < Q THEN 1200
1310 IF T = 0 THEN RETURN
1320 GOTO 1010
```

---

A subroutine in BASIC to bubblesort an array, B, of 100 items.



## Compound Interest Isn't Simple

You buy a car and, after negotiating the trade-in and down payment and fees, you find that there is still \$5000 to pay. At the present time, money can be borrowed at 1.25% per month on the unpaid balance. Or, money can be lent (e.g., deposited) at 9% compounded quarterly.

If the money is borrowed, say to be repaid in monthly installments for 4 years, the monthly payments will be \$139.15, and the total repayment will be \$6679.20. That figure of \$139.15 is obtained from the formula:

$$\frac{I \cdot (1 + I)^n}{(1 + I)^n - 1}$$

where  $I$  is the interest rate per period (e.g., 1.25% per month), and  $n$  is the number of periods (e.g., 48 months).

Or, the purchaser could pay the \$5000 out of savings, and then pay himself \$139.15 per month for 48 months. At the 9% rate, this procedure would build up an amount of \$10578.40. On the other hand, if the \$5000 had been left in the savings account, it would have built up to \$7138.11 at the end of 4 years.

Now, all of the above can be calculated easily with a pocket calculator and perhaps a good compound interest table, such as:

### Financial Compound Interest and Annuity Tables

Financial Publishing Company, Boston

Publication Number 176.

But the true situation is much more complex. If one borrows money, then under current U.S. tax laws, the interest paid is a deduction and, depending on one's "tax bracket," the cost of borrowing is alleviated by this tax break.

Contrariwise, if money is lent, then the tax laws consider the interest that is earned as income, and the gain is offset by a tax bite, whose size is again a function of the "tax bracket."

The term "tax bracket" is one of those locutions that everyone uses; that everyone defines to himself; and that no two people agree on. The simplest definition is this: if a person reported one more dollar of gross earnings, then the part of that dollar that is taken by Internal Revenue is his tax bracket. But even that is oversimplified. It is quite possible that one extra dollar would move a person into the next higher bracket of the IRS's tables, and hence be far more costly than one dollar earlier or later.

Leaving out such pathological cases, we still have:

If \$1000 is <u>paid out</u> in interest on a loan, there is a tax credit so that the \$1000 really costs X dollars less.	If \$1000 is <u>earned</u> in interest on a loan, there is a tax cost so that the \$1000 really earns Y dollars less.
---	---

To determine the actual values of X and Y in any specific case, it would be necessary to calculate the entire Form 1040 both ways. However, as a rough guess (for middle-income people; that is, excluding the very rich and the very poor), X and Y do not differ significantly. Thus, if one has determined his tax bracket (as defined earlier) to be 20%, then either X or Y can be figured to be \$200 without serious error.

So it would seem that the task of calculating the actual cost of the car loan is an excellent computer problem. There are these parameters:

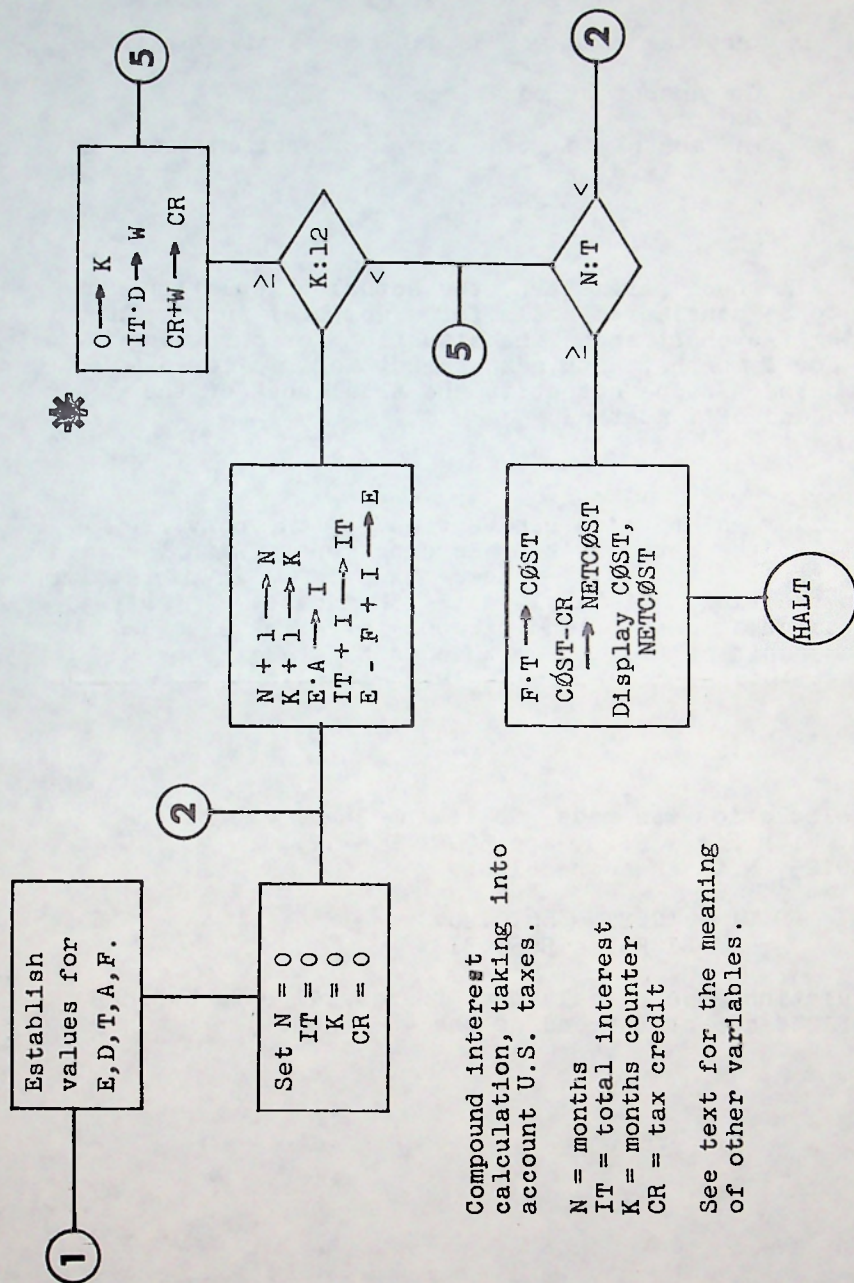
T: The number of months of the loan.

A: The monthly rate for borrowing money (e.g., 1.25% on the unpaid balance).

B: Annual rate for lending money (e.g., 9%).

C: Compounding period for lending (e.g., quarterly).





Compound interest calculation, taking into account U.S. taxes.

N = months  
IT = total interest  
K = months counter  
CR = tax credit

See text for the meaning of other variables.

D: The "tax bracket" as defined earlier.

E: The amount of money needed.

F: The monthly payment for an amortized loan.

With all these parameters, the actual calculation for each case to be considered calls for a computer run. The accompanying flowchart shows the situation for borrowing E dollars for T months, with a tax credit of D allowed every 12 months. The output is the total cost of the loan at the end of T months.

A similar calculation can be made for the converse case (that is, the net result of lending money, with its associated penalty in yearly income taxes). In particular, the box on the flowchart labelled (\*) then has (-W) for (W). Also, the calculation for lending money might have to be made in some unit of time other than months (e.g. some savings institutions advertise daily compounding).

A calculation was made, following the logic of the flowchart, with  $F = 139.15$ ,  $E = 5000$ ,  $D = .2$ ,  $T = 48$ , and  $A = .0125$ , with these results:

CØST = 6679.20  
NETCØST = 5659.31

(The calculation also reveals that there will be a balloon payment of 24¢ due at the end of the 48 months.)





# Schwartz on Powers of 2

In issue number 83 we gave the single-digit distribution of the 6021 digits of the 20000th power of 2, and said "...computer runs indicate that for all  $X > 169$ , every value of

$$F = 2^X$$

contains all the decimal digits 0 through 9. This is certainly a plausible notion, and it has been verified for all values of  $X$  up to 20000."

Dr. Mordecai Schwartz writes:

"The question arises as to just how plausible. Can we quantify it? Can we set an upper limit to the probability  $p$  of ultimately finding an  $F$  which lacks one of the decimal digits 0, 1, ..., 9 as  $X$  extends to infinity? The problem boils down to whether we can sum the infinite sequence of probabilities for all integers  $X > 20,000$ .

For all we know a priori, the gradually increasing probability

$$p_n \rightarrow p \text{ as } n \rightarrow \infty$$

may behave much like the harmonic series which very very slowly increases beyond all bounds if only we take enough terms. Here "beyond all bounds" is meant literally; there is no a priori probability cutoff--not even one, certainly! This is so because the probabilities for specific  $X$ 's are not mutually exclusive. The  $k$  digits of

$$F = 2^X \text{ might be the first } k \text{ digits of } 2^Y$$

where  $Y > X$ . Thus, if we sum individual probabilities for all  $X > 20,000$ , this sum is an upper bound for the true probability  $p$ .

You point out that for  $X = 20,000$ ,  $F$  contains 6021 (decimal) digits. Now observe that

$$2^{(X+3)} = 8 \cdot 2^X$$

usually has one more decimal digit than  $2^X$ , while

$$2^{(X+4)} = 16 \cdot 2^X$$

always does. Then consider the infinite sequence:

(1)  $2^{20001}, 2^{20002}, 2^{20003}, 2^{20004}, 2^{20005}, \dots$

and group the terms in sets of four as shown.

Assume that each term of the first set has 6021 digits, and that each succeeding set has one additional digit. This is another worst-case assumption to maximize the probability of an absent decimal digit, since the calculated probability for any term is greater with fewer digits.

Now, if  $2^X$  has  $k$  integer (digit) positions when expressed in base 10, then a given decimal digit, say 3, has probability

$$(9/10)^k$$

of being absent from every position (we assume random distribution of digits). If we want the probability that some decimal digit--we don't care which--is absent, that probability is

$$10 \cdot (9/10)^k.$$

We are now ready to sum the probability that some digit is absent from at least one term of sequence (1); it is:

$$p \leq [10 \cdot (9/10)^{6021} \cdot 4] + [10 \cdot (9/10)^{6022} \cdot 4] + [10 \cdot (9/10)^{6023} \cdot 4] + \dots$$

$$\text{or, } p < 40 \cdot (9/10) [1 + 9/10 + (9/10)^2 + (9/10)^3 + \dots]$$

The expression in brackets in the last relationship is a geometric progression for which the sum

$$S = \frac{a}{(1 - r)} = \frac{1}{(1 - 9/10)} = 10$$

so that

$$p < 400 \cdot (9/10)^{6021}$$

This gives a numerical upper bound for the probability we are seeking. Using logarithms, it figures out to:



$$p < 10^{-273}$$

This inconceivably minute probability is an upper bound for the chance of ultimately finding a power of 2 that lacks a decimal digit. Any further empirical study of the function that extends the maximum X for which

$$F = 2^X$$

is proven to contain all the decimal digits will diminish our probability p still further.

The only consideration that might invalidate this analysis would be a demonstration that the decimal digits of  $2^X$  as X increases possess some property that negates the assumption of random distribution of the digits. The distribution of the digits that you gave for  $X = 20,000$  is certainly good evidence that our assumption is valid. Further, probabilities for individual terms of (1) diminish so rapidly, that the total residual probability is little more than that of the first undecided term, or at the least, of the same order of magnitude."

To bolster our previous argument a bit further, we ran the 2-digit distribution of the digits of the 20000th power of 2, followed by the same distribution for the 50000th power. It seems fair to state that Dr. Schwartz's assumption of randomness in these numbers has not been vitiated.

The serial (2-digit)  
distribution of the  
6021 digits of the  
20000th power of 2.

00	64	20	63	40	74	60	65	80	62
01	63	21	71	41	61	61	66	81	58
02	54	22	55	42	62	62	53	82	68
03	70	23	64	43	69	63	64	83	60
04	68	24	71	44	51	64	51	84	58
05	57	25	53	45	61	65	62	85	55
06	61	26	50	46	55	66	59	86	55
07	61	27	62	47	45	67	47	87	62
08	61	28	51	48	55	68	60	88	61
09	55	29	43	49	56	69	58	89	71
10	60	30	64	50	55	70	53	90	54
11	55	31	70	51	49	71	67	91	55
12	55	32	63	52	60	72	56	92	57
13	66	33	55	53	57	73	56	93	60
14	58	34	66	54	57	74	59	94	50
15	74	35	61	55	57	75	69	95	49
16	59	36	62	56	57	76	67	96	61
17	65	37	65	57	62	77	69	97	77
18	59	38	57	58	78	78	67	98	61
19	64	39	59	59	66	79	52	99	65

1518	1552	1505	1482	1475	1506	1510	1469	1511	1524
0	1	2	3	4	5	6	7	8	9

The frequency (1-digit) distribution of the 15052 digits of the 50000th power of 2.

00	135	20	148	40	166	60	159	80	163
01	159	21	149	41	163	61	177	81	149
02	154	22	152	42	149	62	150	82	171
03	146	23	165	43	129	63	168	83	140
04	142	24	153	44	139	64	137	84	145
05	179	25	164	45	156	65	135	85	148
06	140	26	142	46	137	66	138	86	158
07	128	27	143	47	161	67	136	87	154
08	152	28	158	48	130	68	154	88	138
09	183	29	131	49	145	69	155	89	145
10	150	30	139	50	154	70	156	90	148
11	173	31	145	51	162	71	133	91	142
12	141	32	150	52	159	72	120	92	159
13	160	33	129	53	154	73	140	93	150
14	147	34	156	54	137	74	155	94	164
15	145	35	144	55	152	75	130	95	153
16	169	36	153	56	144	76	173	96	156
17	150	37	154	57	147	77	152	97	144
18	173	38	155	58	156	78	146	98	149
19	144	39	157	59	141	79	164	99	160

The serial (2-digit) distribution of the 15052 digits of the 50000th power of 2.

## Answer to Last Month's Challenge Problem:

The man in the rowboat hears the shot 48.7 seconds after the man in the tree. The position of the airplane at the time of the shot has no bearing on the problem, nor does the difference in elevation of the three participants.



# The Six-Digit Algorithm

7 2 1	9 3 6
6 7 4	8 5 6
5 7 6	9 4 4
5 4 3	7 4 4
4 0 3	9 9 2
3 9 9	7 7 6
3 0 9	6 2 4
1 9 2	8 1 6
1 5 6	6 7 2
1 0 4	8 3 2
8 6	5 2 8
4 5	4 0 8
1 8	3 6 0
6	4 8 0
2	8 8 0
1	7 6 0
	7 6 0
	0

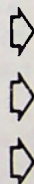
1  
2  
3  
4  
5  
6  
7  
8  
9  
10  
11  
12  
13  
14  
15  
16  
17

Consider the number 721936. If its two halves (721 and 936) are multiplied together, the result is 674856, as shown in step 1 in the accompanying diagram.

The process repeats... Specifically, the low-order three digits are multiplied by the balance of the number to produce the next number in the sequence. For the starting number 721936, the process terminates with zero at the 17th stage.

The number of stages that any given 6-digit number will take to complete this process is somewhat unpredictable. The Figure on page 20 shows two consecutive numbers; one of them goes out in 6 stages; the other in 10 stages.

The process suggests some Problems that can be solved with any computer:



(Problems connected with the 6-digit algorithm)

A. Are there 6-digit numbers that can survive for more than 17 stages?

B. Considering 100000 as the smallest 6-digit number, there are 899999 numbers to be examined. Many of them go out in just 2 stages (e.g., 100002). What is the distribution of the results for the 899999 cases?

C. What is the most probable number of stages for a random 6-digit number to take in this process?

D. The same algorithm can be applied to 8-digit numbers (breaking each product into two 4-digit factors, and so on). What is the greatest number of stages that an 8-digit number can take?

E. For the 8-digit case, there are 89,999,999 numbers to be considered. What is the distribution of the results for all those cases?



PROBLEM 280

922	983	922	984	
906	326	907	248	1
295	356	224	936	2
105	020	209	664	3
2	100	138	776	4
	200	107	088	5
	0	9	416	6
		3	744	7
		2	232	8
			464	9
			0	10

The 6-digit algorithm illustrated for two consecutive numbers.